Project 3 in Udacity Reinformcement Learning Nanodegree

Project report

.

Introduction

In this project, we are working with Tennis environment, where two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

The task is episodic, and to solve the environment, the agents must get an average score of +0.5 over 100 consecutive episodes. In each episode, we add up the rewards for each agent and get a final score for each agent. This yields 2 scores, maximum of which will be consider as the result of the episode. The environment is considered solved, when the average over 100 episodes is at least +0.5.

Solution method

Multi Agent Deep Deterministic Policy Gradient is used to solve this environment. Each agent has two neural networks, one as actor and one as critic. The actor network has the state vector as input and action vector as output. Critic network has both state vector and action vector as inputs and estimates the reward. The networks used for the agents are exactly the ones used in Project 2 Continuous Control with the inputs and outputs adjusted to the current environment. The input of the actor network is the state vector with 24 states and the output will be two actions. The critic network has the state vector as input and at the second layer, the two actions are added. The output of the critic network is one value.

DDPG

In DDPG, two neural networks are used, one as actor and one as critic. The actor network produces the believed best action for a given state description, which is basically the deterministic policy, by learning $\mu(s;\theta_\mu) = argmax_a Q(s,a)$, with critic, on the other hand, evaluates the action value function, for the best believed action, i.e. $Q(s,\mu(s,\theta_\mu);\theta_Q)$. Here, s and a denote environment state and actions and θ the approximation parameters, or the network weights.

During training, the networks are updated in the following way:

1. Given the current state (s), the actor network suggests actions ($a = \mu(s)$).
2. The actions are sent to the environment and new states (s_{next}) and rewards (r) are recorded.
3. New actions are calculaed for next state by the actor, i.e. $a_{next} = \mu(s_{next})$.
4. A new value for the current state is calculated by adding the obtained reward and discounted estimated reward for the next state, i.e. $Q = r + \gamma \times Q(s_{next}, \mu(s_{next}))$, where γ is the discount factor.
5. The critic network is updated to minimize the difference between the difference between the current estimate, $Q(s, a)$ and the above calculated discounted reward.
6. The actor network will be updated so that the reward, predicted by the critic network, will be maximized. That is done by first using the actor network to get the suggested actions and then the critic to estimate the value from the states and believed best action. That means, we maximize $Q(s, \mu(s))$

DDPG, is using two networks, local and target. During the learning process for the critic network, the target actor is used to evaluate the actions for the next state and the weights of the local critic network are updated. Similarly, the weights of the local actor network are updated using the local critic network with upated weights. A scheme called soft update to update the target network, i.e. $\theta_{target} = \tau \times \theta_{local} + (1 - \tau) \times \theta_{target}$, where τ is a relaxation parameter that hould be set to a small value.

Networks with two hidden layes with 128 neurons were used as shown below:

```
Actor network:  Actor(
  (fc1): Linear(in_features=24, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=2, bias=True)
)
Critic network:  Critic(
  (fcs1): Linear(in_features=24, out_features=128, bias=True)
  (fc2): Linear(in_features=130, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=1, bias=True)
)
```

Experience Replay with and without Prioritization

Implementation of experience replay and the prioritized version is similar to Project 1 Navigation. For experience replay, two approaches are applied. The two agent can have their own buffer or share the buffer. The latter is possible as the agent experience the world in the same way and therefore can share the experiences. This would potentially lead to a richer data for learning.

When the prioritization is added to the selection of the batch from the replay memory, the probability of selecting a tuple is related to the error associated to that tuple. To do that, first the recorded errors are collected and scaled, so that they can be used as a probability for the selection. Then, the index of the selected tupples is saved in a property of the buffer of type `PrioritizedReplayBuffer` , called `self.indx` of type `np.array(batch_size, dtype=int)`

```
losses = np.array([e.loss for e in self.memory])
        self.max_loss = losses.max()
        losses = (losses + self.MIN_PROP)**(self.alpha)
        loss_sum = losses.sum()
        p = losses/loss_sum

self.indx=np.random.choice(list(range(len(self.memory))),self.batch_size, p=p,
replace=False)
```

The error is taken from the actor network and the values in the buffer are updated as follows:

```
# Compute actor loss
        actions_pred = self.actor_local(states)
        actor_losses = -self.critic_local(states, actions_pred)
        self.memory.update_losses(i_episode,
abs(actor_losses).cpu().data.numpy())

        actor_loss = (actor_losses* weights).mean()
        # Minimize the loss
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()
```

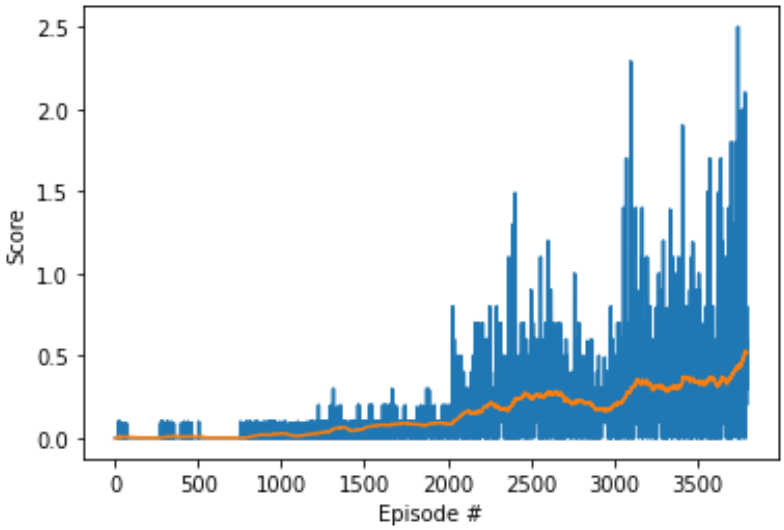
and to obtain that the implementation should be changed slightly. After that the expected reward and the estimate of the target reward are calculated, the losses are computed as the difference between them. These losses are saved for later use and the total error is calculated for minimization of the error.

The prioritized experience replay involves two parameters α and β. α is used in canlculation of probability of choosing a tuple from the memory, i.e. $P_i = \epsilon_i^\alpha / \Sigma \epsilon_i^\alpha$, where ϵ_i is the latest error for the ith record in the memory. β is a parameter for the weight calculation is the contribution to the total error from each case in the batch. That is $w_i = (P_i N)^{-\beta}$/Max_i(w_i). Note that if $\alpha = 0$ the batch is randomly sampled and if $\beta = 0$ the weights are equal. α was set to 0.7 and β started at 0.4 and increased linerly to 1 over 1000 episodes.

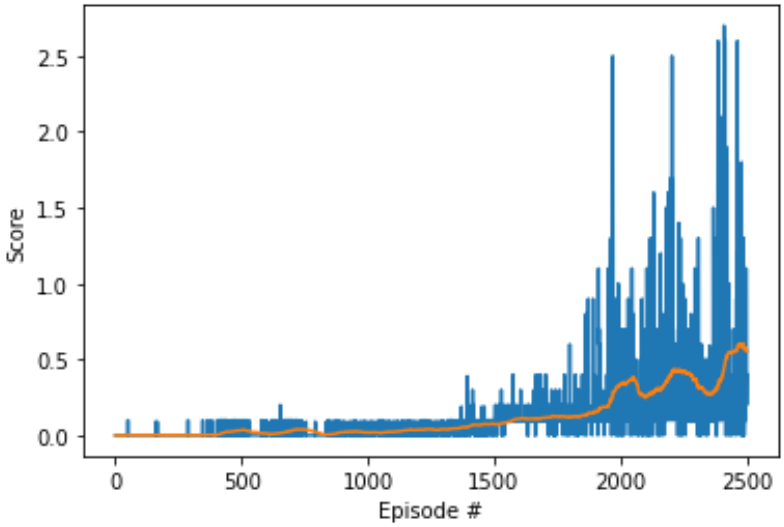
Studied cases and results

The first network selection presented above, worked well. Therefore, I decided to keep them and instead see how experience sharing and prioritisation can affect the learning process. First, four cases were considered, with and without sharing and with or without prioritization. The results are shown below:

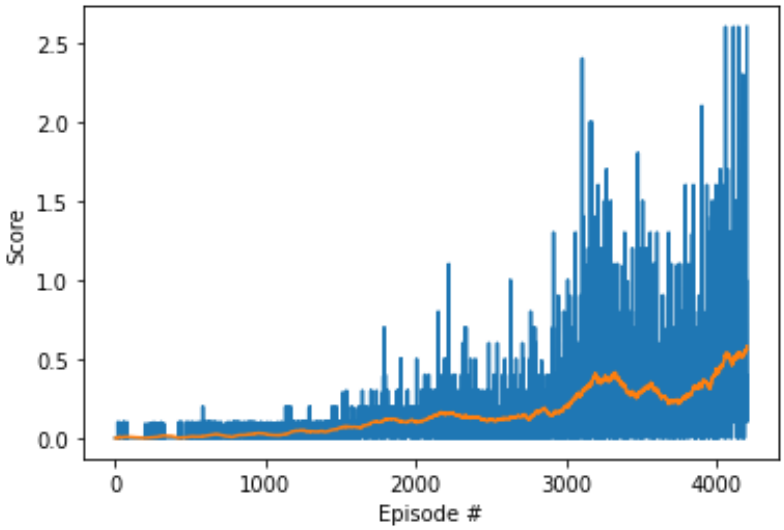
Separate buffer for each agent and without prioritization



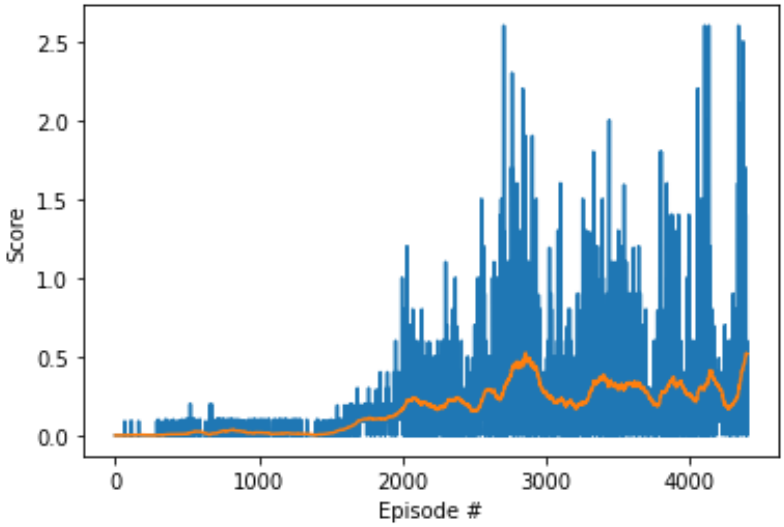
Shared buffer between the agents and without prioritization



Separate buffer for each agent and with prioritization



Shared buffer between the agents and with prioritization



As can be found in the above presentation of results, the best result has been achieved with sharing experience between the agents. Somewhat unexpectedly, I found that prioritization did not help and in addition the process was more expensive. A speculation could be that it is because of the simplicity of scoring scheme and that the agent learns to survive without needs for special tricks.

In this environment, the initial data collected has few successful shots. I thought it will be better to have data that also have some good shots. I did it by defining a vector in the constructor of each agent that records what type of reward it has obtained.

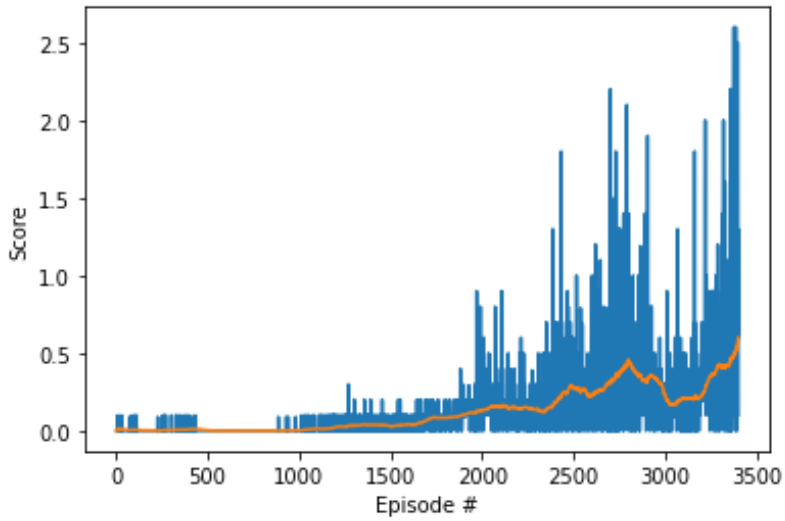
```
self.short_memory = deque(maxlen=100)       # Saves statistics about the 10
last inputs
```

and then use it in the step method to control what data comes it.

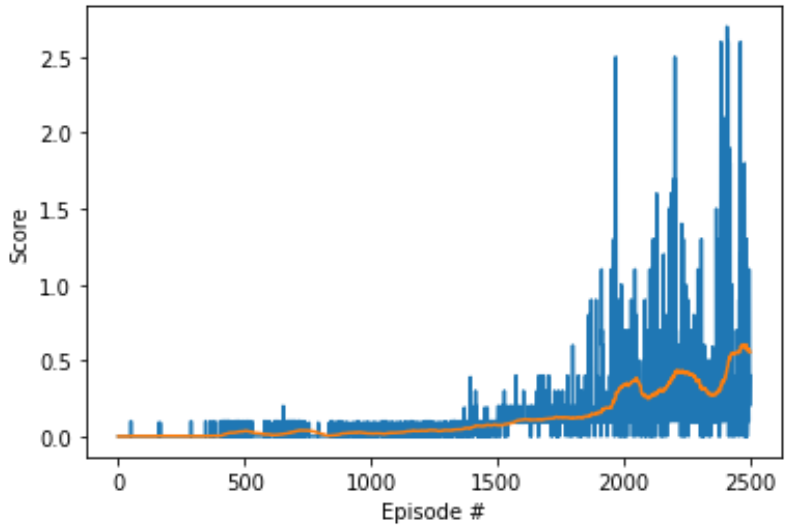
```
if ((sum(self.short_memory) > 0) or (reward > 0)):
            self.short_memory.append(1 if reward > 0 else 0)
            self.memory.add(state, action, reward, next_state, done)
```

Having a vector length of 100, requiring a positive sum means demanding that 1% of the shots should be a good one. Below figures show the results of implementation of this selective memory function. As can be seen here, compared to the case with shared memory, the target is achieved later. My interpretation is that the fact that the data collection slows down because of this change causes this slower convergence.

Selective shared buffer between the agents, without prioritization

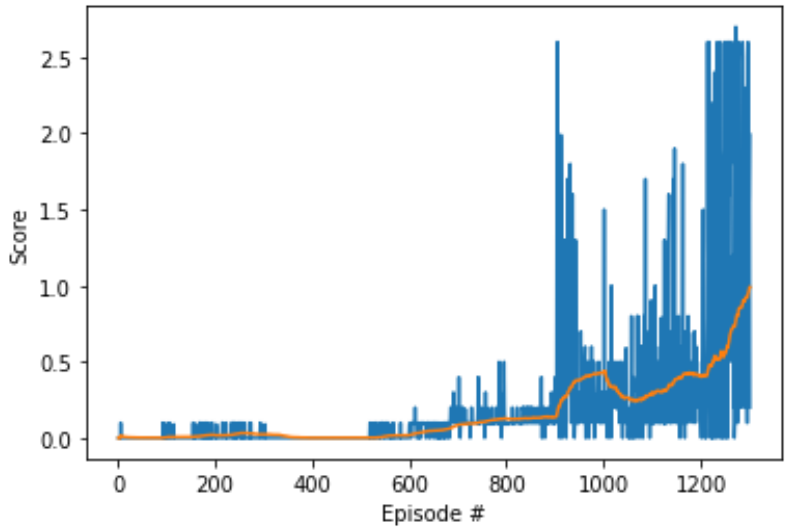


Shared buffer between the agents and without prioritization

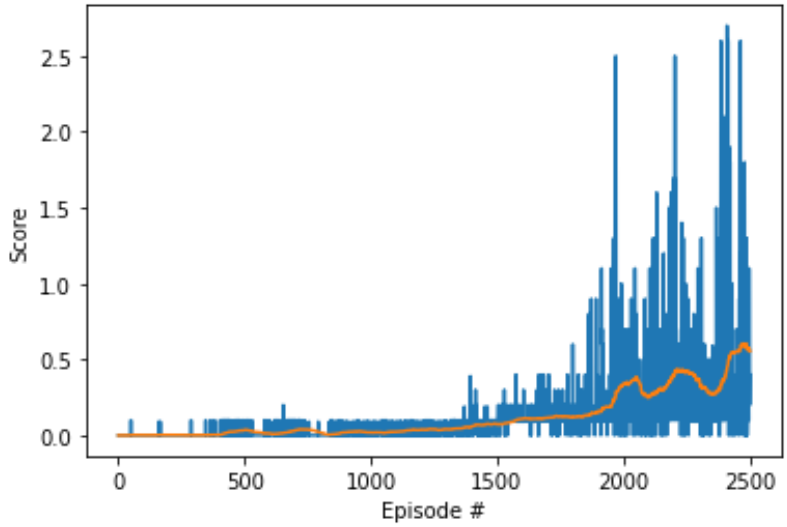


As the final case, I thought as matter of fact we do not need two agents. Once we have a trained agent, it can be used to control both rackets. Below figures and comparisons confirm the hypothesis. The target is reached faster with better result.

One agent learning from both rewards, without prioritization



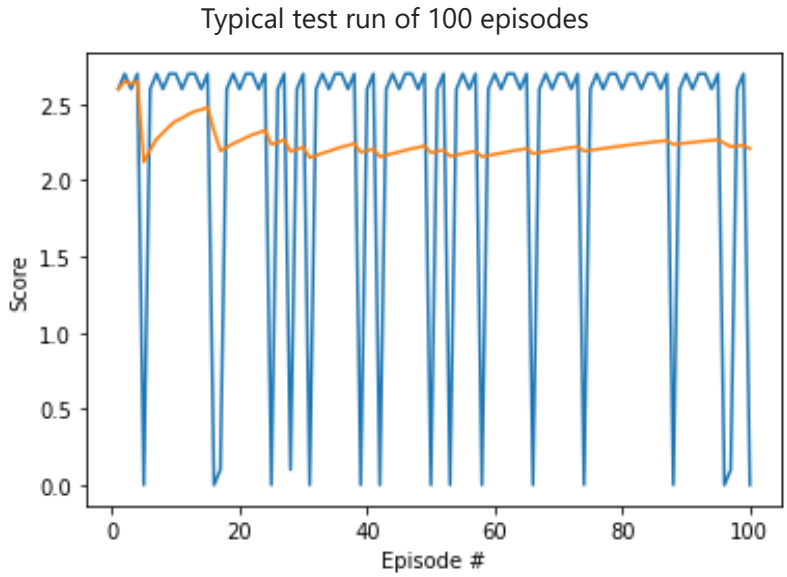
Shared buffer between the agents and without prioritization



Above results can be repeated by running the following notebooks:

1. `Tennis_SimpleMemory.ipynb` solves the environment with two separate agents.
2. `Tennis_SimpleSharedMemory.ipynb` solves the environment with two agents that share memory.
3. `Tennis_PrioritizedExperienceReplay.ipynb` solves the environment with two agents, with prioritized experience replay.
4. `Tennis_PrioritizedExperienceReplayShared.ipynb` solves the environment with two agents, with prioritized experience replay, when they share memory.
5. `Tennis_SelectiveMemory.ipynb` solves the environment with two separate agents, where the memory is selective and requires 1% of good shots.
6. `Tennis_SimpleMemorySingleAgent.ipynb` solves the environment with one single agent.

If you choose to test, you should have a result similar to the figure below:



Summary, conclusion and discussion

Multi Agent implementation of Deep Deterministic Policy Gradient (MADDPG) was used to solve the tennis environment. The attention was paid to use of data, sharing between the agent and prioritization. All variants tested were able to solve the problem but with variable effort (number of episodes to reach the target).

The best result was achieved with two agent sharing the expeiences, which reached the target after 2500 episodes. When they did not shared experiences, the target was reached after 3800 episodes. One single agent using experiences from both rackets in the environment could reach the target after 1300 episodes.

With the background from the previous two projects, solving this enviroinment with multi agent appeared quite straight forward once I understood the problem and the path to solution. Also a substantial part of the code could be reused from those two labs.

Multi Agent implementation of Deep Deterministic Policy Gradient is simple and works well. Quality and volume of available data plays obviously an important role. An important point to mention is that the nature of problem must be understood well and used to make the best use of the available data.

The main weakness I find with the current implementation is that not much is happening during the 1st period of time. For improving the performance, I would work on the data, both quality and volume. A suggestion could be to collect data, without learning and have a data set that has a good combition of good and bad cases. Further, initial immitation and learning from a human would make the learning process faster.